EPISODIO 3

946

Después de que Jacu investigara el servidor de la empresa Capsule Corp con la ayuda del Dr.Brief, las sospechas de Trunks quedaron confirmadas. La existencia de un fichero sospechoso que no ha sido creado por nadie de la empresa, parece indicar que ésta ha sido comprometida. Para aclarar el enigma, el Dr.Brief pide a Bulma que estudie el fichero en cuestión y extraiga cualquier información relevante que pudiera arrojar luz sobre el caso. Bulma consigue obtener la información de su creador, el Dr. Raichi, y descubre, además, que el contenido del fichero está escrito en una extraña lengua de la raza Tsufur. Para descifrar el contenido y obtener el texto en un lenguaje que ellos comprendan necesitan una clave. ¿Podrás encontrarla?

Descarga del fichero: https://drive.google.com/file/d/1UihvI5nEjkarfM03DV8J5RgJ2ZD1zy_T/view?usp=sharing

Info: La flag tiene el formato UAM{md5 del string encontrado}

Descargamos el binario, “main”, nada más lanzarlos nos pide una flag. Metemos un texto, y nos devuelve Flag incorrecta:

```
nacho@kali:~/UAM/201910-BolaDrac3$ ./main
Flag: holacaracola
Flag incorrecta!! D:
```

Analizamos el fichero y nos dice que es un ejecutable de 64 bits:

```
armitage
:~/UAM/201910-BolaDrac3$ file main
main: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64
/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=6c62b346db422600ce26c67127d49dbb77ba487
8, not stripped
```

Lo abrimos con IDA, y analizamos la función “main”, vemos que solicita por entrada estándar 0x14h caracteres:

```
lea     rax, [rbp+s]
mov     esi, 14h        ; n
mov     rdi, rax        ; s
call    _fgets
```

Tenemos también una constante que mete, será posiblemente la clave del cifrado / descifrado.

```
mov     [rbp+inicio_constante], 7
mov     [rbp+var_3B], 59h
mov     [rbp+var_3A], 1Dh
mov     [rbp+var_39], 36h
mov     [rbp+var_38], 1Ah
mov     [rbp+var_37], 59h
mov     [rbp+var_36], 36h
mov     [rbp+var_35], 5Ah
mov     [rbp+var_34], 5Dh
mov     [rbp+var_33], 1Ah
mov     [rbp+var_32], 30h
mov     [rbp+var_31], 0
```

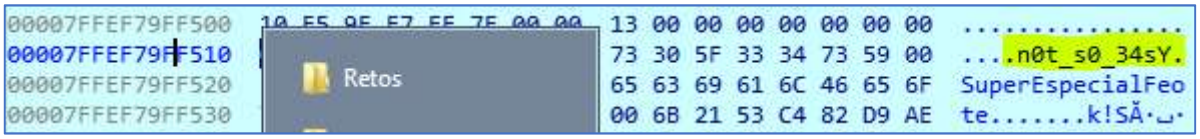
Ahora llama a una función de descifrado, no le pasa nuestra cadena de entrada, solamente le pasa la constante:

```
lea     rax, [rbp+inicio_constante]
mov     rdi, rax    ; s
call    _strlen
mov     rdx, rax
lea     rax, [rbp+inicio_constante]
mov     rsi, rdx    ; unsigned __int64
mov     rdi, rax    ; char *
call    _Z7decryptPcm ; decrypt(char *,ulong)
```

En la función únicamente recorre cada carácter de la cadena, y le hace un XOR con 0x69h:

```
mov     eax, [rbp+indice_bucle]
movsxd  rdx, eax
mov     rax, [rbp+cadena_constante]
add     rax, rdx
movzx   ecx, byte ptr [rax]
mov     eax, [rbp+indice_bucle]
movsxd  rdx, eax
mov     rax, [rbp+cadena_constante]
add     rax, rdx
xor     ecx, 69h
mov     edx, ecx
mov     [rax], dl
add     [rbp+indice_bucle], 1
jmp     short loc_400BFD
```

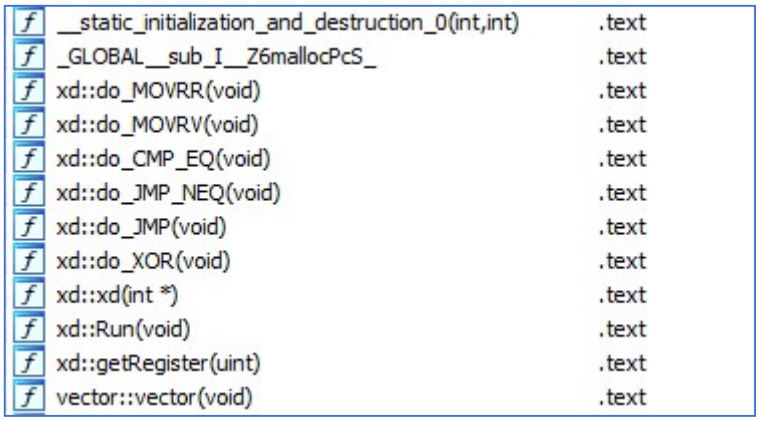
Devolviendo la cadena resultante:



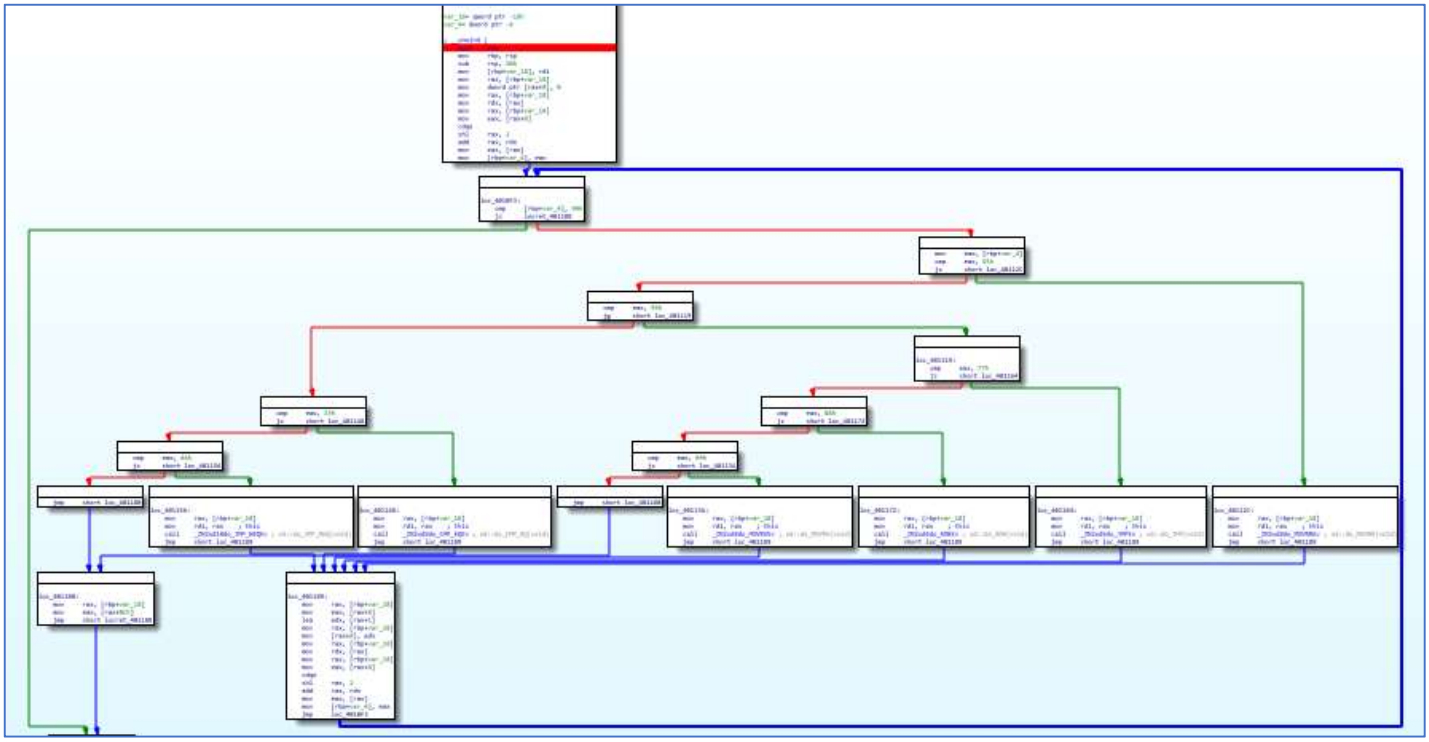
La cadena es: n0t_s0_34sY, si probamos a meterla directamente en la entrada:

Que no es la Flag correcta, así que debemos mirar más a fondo. Además, este resultado nos ha dado haciendo debug desde IDA, si probamos lo mismo directamente desde el ejecutable… nos da flag incorrecta. Algo está diferente en caso de debug o de ejecución directa.

Analizando con IDA, vemos que tiene una serie de funciones, que no hemos usado, y que parecen como si el binario fuera una máquina virtual:



Vemos como en estas funciones se ejecuta código interesante para analizar. La función xd::Run es el claro tipo de un “switch”, que además según cada valor va llamando a las funciones “do_MOVRR”, “do_COM_EQ”, etc.. lo cual nos reafirma en que es una máquina virtual, y según los valores de entrada va llamando a los distintos opcodes de ejecución:



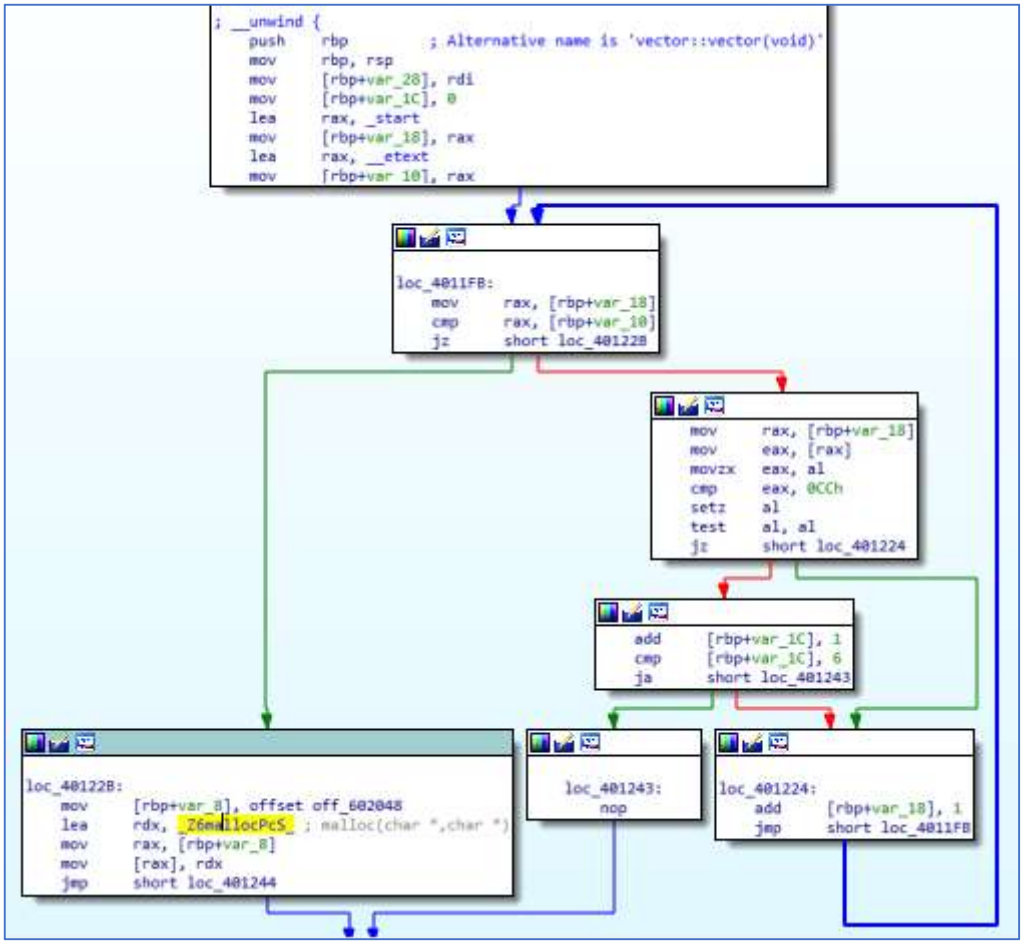
Busco más referencias a la función gets, o a la función printf, debería de usarse en algún otro sitio si finalmente se pinta una flag en otra parte del código, o se toma un valor de entrada, pero no existen. Los únicos sitios donde se usan gets y printf es en la función main, que ya he usado para sacar la flag “correcta”, pero no verdadera.

Busco referencias a estas funciones anteriores desde la función main, pero no existen, no encuentro la forma de invocar a dichas funciones desde el código principal. Todas estas funciones “xd” se invocan entre sí de una manera lógica, y la madre de todas sería “_GLOBAL__sub_I__Z6mallocPcS_”, que ya no tiene referencias directas, salvo en una variable que apunta a su dirección.

Podría ser que se estuvieran ejecutando antes, o de alguna manera que no me diera cuenta, así que pongo un breakpoint en cada una de ellas, y lo ejecuto. Para ello, como siempre, desde la MV Kali lanzamos el “linuxserver64”, y desde la MV Windows donde tengo el IDA, configuro el debugger remoto apuntando a la IP de la otra VM. Bingo! Vemos que se para en la función inicial antes de llegar a esa función main:

```
_GLOBAL__sub_I__Z6mallocPcS_ proc near
; __unwind {
push    rbp
mov     rbp, rsp
mov     esi, 0FFFFh     ; int
mov     edi, 1          ; int
call    _Z41__static_initialization_and_destruction_0ii ; __static_initialization_and_destruction_0(int,int)
pop     rbp
retn
```

Si seguimos la ejecución, vemos como llega a la función vector::vector, que incluye este bucle:



Analizando su contenido, vemos inicia un contador con el valor del offset de memoria de la función “start” y lo va comparando hasta el offset de memoria de la variable “_etext”. Dentro tiene dos bucles, uno que va de 0 a 0xCC, y el otro de 0 a 6, y así va subiendo de uno en uno el valor inicial de start. En algún momento la condición del bucle se debería cumplir, y saldría por la función de abajo izquierda que invoca el método “malloc (char *, char*) que es el que necesitamos ejecutar.

Sin embargo, nunca llega a ejecutarse ese código malloc. El total de pasos del bucle es demasiado corto, y no llega a satisfacer la condición final de que los offset de start y de _etext se iguales. Es posible que esto fuera la diferencia que veíamos entre ejecutarlo directamente desde el Shell, o a través del Debug. No pasa nada, esto nos lo podemos saltar. Para ello vamos a modificar a mano, en debug, el valor de la flag Z y que pase por el trozo de código que queremos.

Cuando llegamos al trozo de código que lleva el malloc, vemos que no lo está ejecutando, sino que lo que hace es asignar el offset de la dirección malloc sobre el offset 0x602048. Si vemos lo que hay en ese offset, vemos que es la tabla .GOT.PLT, y ese offset apunta a la función strcmp, de una manera dinámica que el propio ejecutable asignará al cargarse para permitir el acceso a funciones de librerías externas.

```
.got.plt:0000000000602030 E6 05 40 00 00 00 00 00 off_602030 dq offset loc_4005E6        ; DATA XREF: _printf↑r
.got.plt:0000000000602038 F6 05 40 00 00 00 00 00 off_602038 dq offset loc_4005F6        ; DATA XREF: _memset↑r
.got.plt:0000000000602040 06 06 40 00 00 00 00 00 off_602040 dq offset loc_400606        ; DATA XREF: _fgets↑r
.got.plt:0000000000602048 16 06 40 00 00 00 00 00 off_602048 dq offset loc_400616        ; DATA XREF: _strcmp↑r
```

Si ejecutamos el código, vemos como modifica esa tabla .GOT.PLT, y ahora ya no apunta a la dirección externa de strcmp, sino a la dirección de _Z6mallocPcS, que es dicha función malloc que vimos antes.

```
.got.plt:0000000000602030 E6 05 40 00 00 00 00 00 off_602030 dq offset loc_4005E6        ; DATA XREF: _printf↑r
.got.plt:0000000000602038 F6 05 40 00 00 00 00 00 off_602038 dq offset loc_4005F6        ; DATA XREF: _memset↑r
.got.plt:0000000000602040 06 06 40 00 00 00 00 00 off_602040 dq offset loc_400606        ; DATA XREF: _fgets↑r
.got.plt:0000000000602048 07 07 40 00 00 00 00 00 off_602048 dq offset _Z6mallocPcS_     ; DATA XREF: _strcmp↑r
```

Por tanto, ahora seguirá la ejecución de una manera normal, nos pedirá la flag como hicimos al principio, pero cuando llegue a validar la flag que metimos e invoque a strcmp, estará invocando realmente a la función malloc.

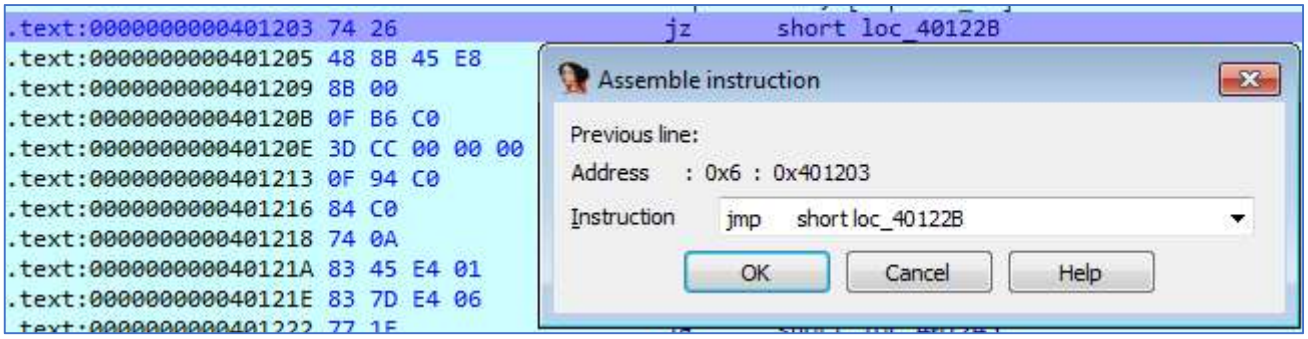
Continuamos con la ejecución, la única condición inicial que debe cumplir la flag que metamos es que tenga el mismo tamaño que la cadena que vimos “n0t_s0_34sY”, es decir, 0xB posiciones, para que entre por esa parte del bucle. Vemos como llega a este trozo de ejecución:

```
48 8D 55 C4             lea     rdx, [rbp+inicio_constante]
48 8D 45 D0             lea     rax, [rbp+cadena_entrada]
48 89 D6                mov     rsi, rdx        ; s2
48 89 C7                mov     rdi, rax        ; s1
E8 E2 F8 FF FF          call    _strcmp
```

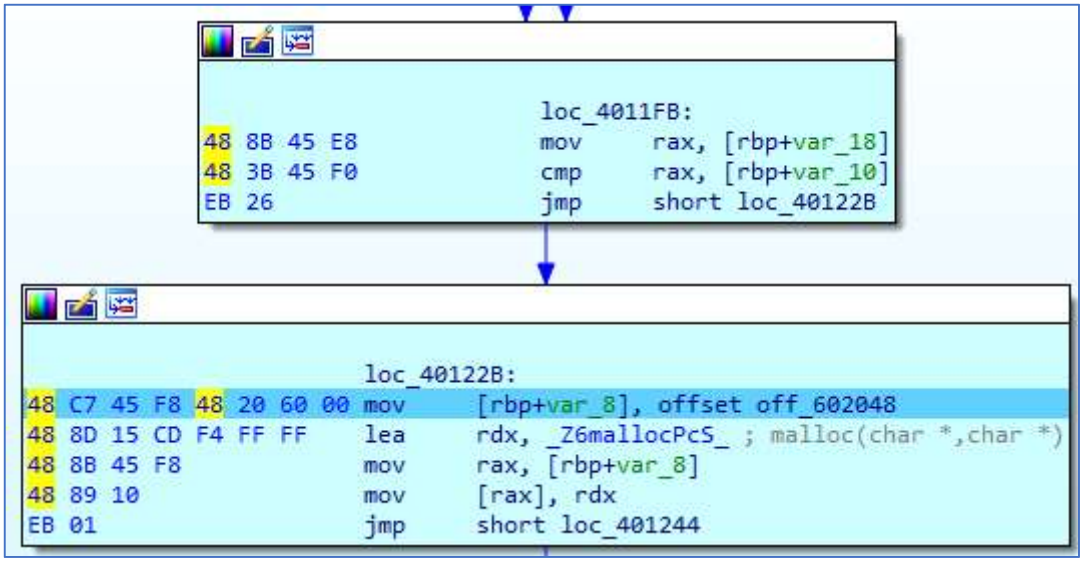
Pero a la hora de ejecutar ese _strcmp.. invoca a malloc, que justo además coincide en que recibe como parámetro dos char*, es decir, dos cadenas:

```
                        ; __unwind {
55                      push    rbp
48 89 E5                mov     rbp, rsp
48 81 EC B0 02 00 00    sub     rsp, 2B0h
48 89 BD 58 FD FF FF    mov     [rbp+s], rdi
48 89 B5 50 FD FF FF    mov     [rbp+var_2B0], rsi
64 48 8B 04 25 28 00 00+mov     rax, fs:28h
48 89 45 F8             mov     [rbp+var_8], rax
31 C0                   xor     eax, eax
48 8B 85 58 FD FF FF    mov     rax, [rbp+s]
48 89 C7                mov     rdi, rax        ; s
E8 82 FE FF FF          call    _strlen
89 85 68 FD FF FF       mov     [rbp+var_298], eax
48 8D 95 D0 FD FF FF    lea     rdx, [rbp+var_230]
```

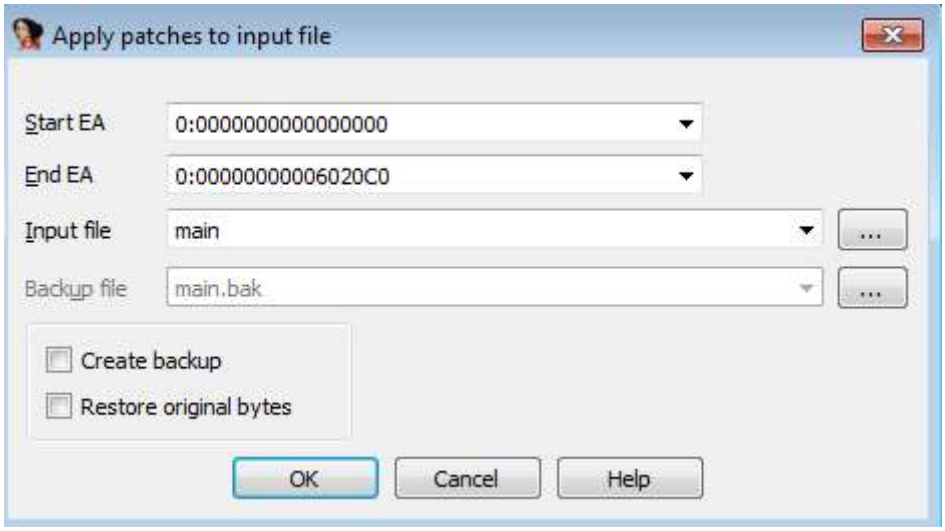
Por tanto, se confirma que debemos hacer que el programa pase por ese malloc. Esta vez lo hemos forzamos temporalmente modificando en debug el valor de la flag Z, es posible que haya alguna manera de forzar a que el programa pase por ahí, pero como no se me ocurre otra ni tengo mucho tiempo para pensar más, y soy un poco bruto a veces, vamos a meterle un “patch” al ejecutable, y modificamos la instrucción de salto condicional que teníamos (JZ), por una con un salto fijo (JMP), para que lo ejecute siempre:



Ahora ya vemos como siempre entra por nuestra parte del bucle interesada, quién dijo miedo!

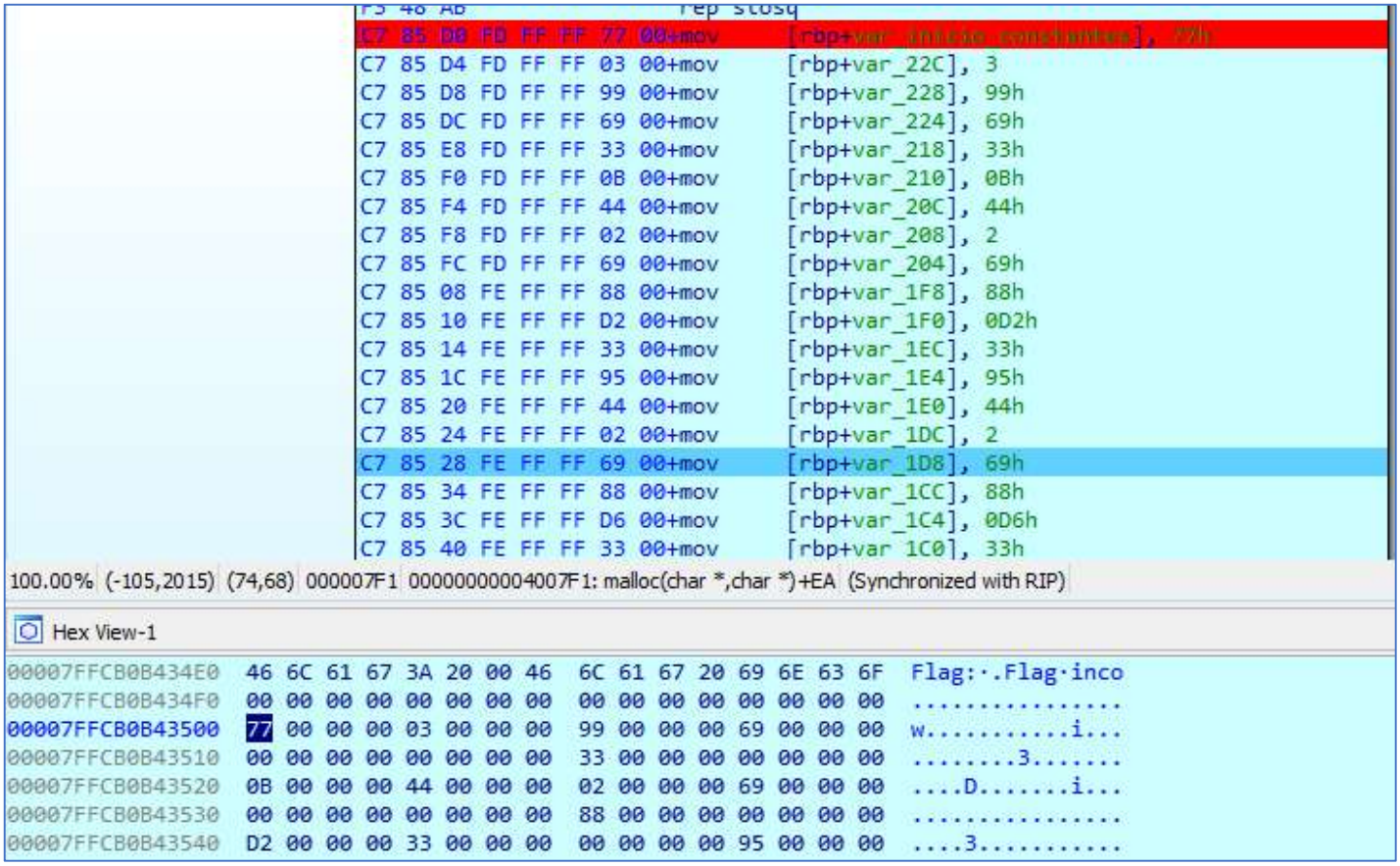


Después hay que grabar los cambios el ejecutable “main”, para que al debugear también coja los cambios.



Ahora ya es cuestión de analizar qué hace esa función malloc y sus hijas, para detectar cual debería ser la flag buena a introducir. De momento solo sabemos que debe tener de tamaño 0xB, vamos a probar con esta e iremos viendo donde almacena esos valores y qué hace con ellos:

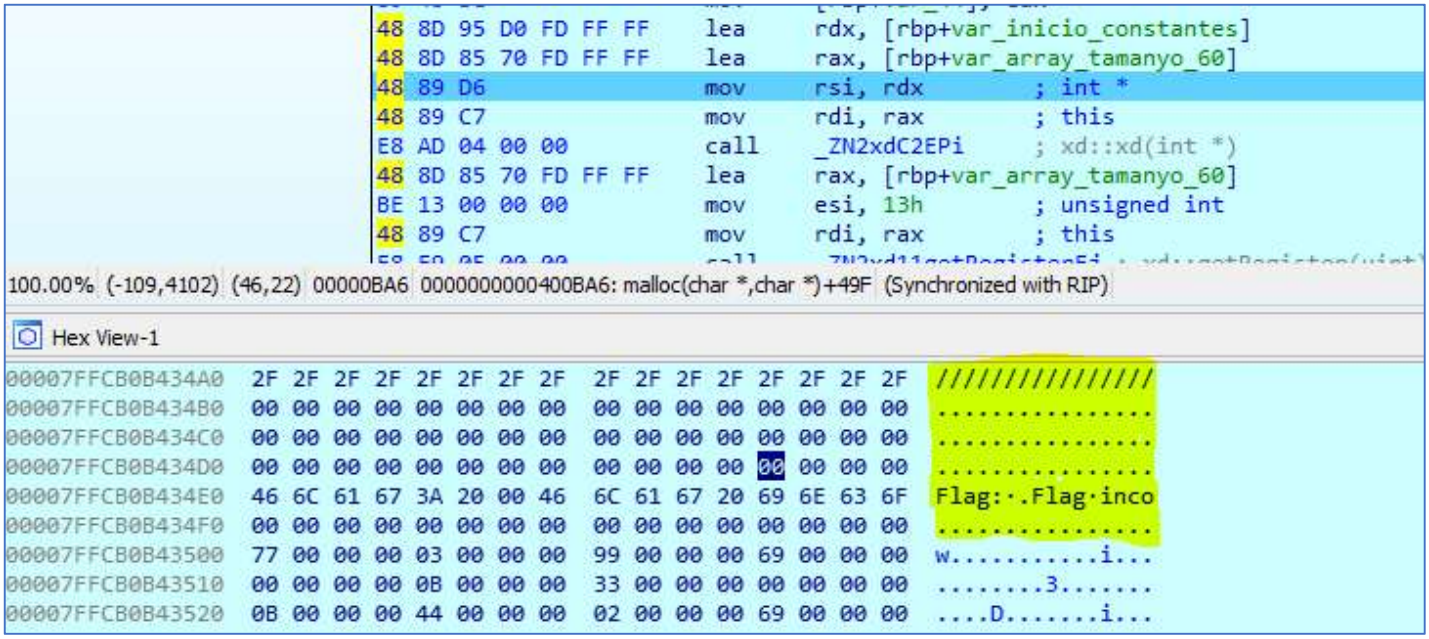
Al inicio de la función malloc, vemos como utiliza una zona de la pila para ir almacenando valores constantes, cada valor (1 byte) lo almacena usando 4 bytes. Esto tiene pinta de ser el código de instrucciones a ejecutar por la máquina virtual:



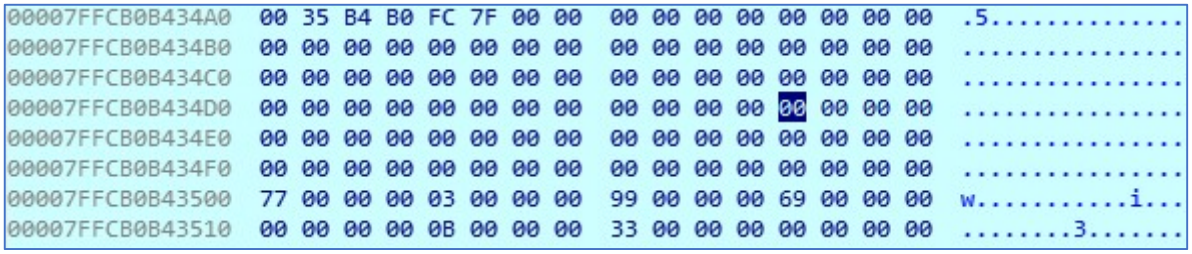
Después, va cogiendo los valores de la cadena que metimos por gets, y los va almacenando combinados con los valores constantes anteriores, quedando la zona de memoria de esta manera. Los valores en amarillo son los valores de la flag que metí, salvo el primero que ahí metió el tamaño de la cadena (0xB):

```
00007FFCB0B43500  77 00 00 00 03 00 00 00  99 00 00 00 69 00 00 00  w...........i...
00007FFCB0B43510  00 00 00 00 0B 00 00 00  33 00 00 00 00 00 00 00  ........3.......
00007FFCB0B43520  0B 00 00 00 44 00 00 00  02 00 00 00 69 00 00 00  ....D.......i...
00007FFCB0B43530  00 00 00 00 62 00 00 00  88 00 00 00 00 00 00 00  ....b...........
00007FFCB0B43540  D2 00 00 00 33 00 00 00  00 00 00 00 95 00 00 00  ....3...........
00007FFCB0B43550  44 00 00 00 02 00 00 00  69 00 00 00 00 00 00 00  D.......i.......
00007FFCB0B43560  61 00 00 00 88 00 00 00  00 00 00 00 D6 00 00 00  a...............
00007FFCB0B43570  33 00 00 00 00 00 00 00  E6 00 00 00 44 00 00 00  3...........D...
00007FFCB0B43580  02 00 00 00 69 00 00 00  00 00 00 00 6E 00 00 00  ....i.......n...
00007FFCB0B43590  88 00 00 00 00 00 00 00  87 00 00 00 33 00 00 00  ............3...
00007FFCB0B435A0  00 00 00 00 D3 00 00 00  44 00 00 00 02 00 00 00  ........D.......
00007FFCB0B435B0  69 00 00 00 00 00 00 00  64 00 00 00 88 00 00 00  i.......d.......
00007FFCB0B435C0  00 00 00 00 EA 00 00 00  33 00 00 00 00 00 00 00  ........3.......
00007FFCB0B435D0  B5 00 00 00 44 00 00 00  02 00 00 00 69 00 00 00  ....D.......i...
00007FFCB0B435E0  00 00 00 00 65 00 00 00  88 00 00 00 00 00 00 00  ....e...........
00007FFCB0B435F0  D4 00 00 00 33 00 00 00  00 00 00 00 BC 00 00 00  ....3...........
00007FFCB0B43600  44 00 00 00 02 00 00 00  69 00 00 00 00 00 00 00  D.......i.......
00007FFCB0B43610  72 00 00 00 88 00 00 00  00 00 00 00 02 00 00 00  r...............
00007FFCB0B43620  33 00 00 00 00 00 00 00  32 00 00 00 44 00 00 00  3.......2...D...
00007FFCB0B43630  02 00 00 00 69 00 00 00  00 00 00 00 61 00 00 00  ....i.......a...
00007FFCB0B43640  88 00 00 00 00 00 00 00  1B 00 00 00 33 00 00 00  ............3...
```

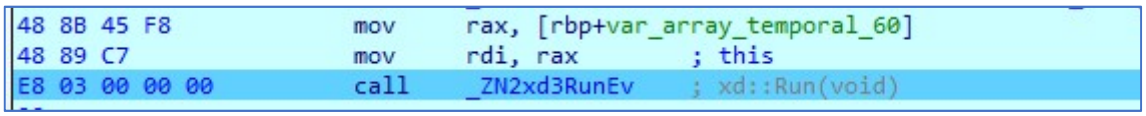
Después define una zona de memoria, justo antes, de 0x60 posiciones (abajo en amarillo), que le pasará a la siguiente función xd::xd, y que veremos que usa como espacio temporal:



Ya dentro de dicha función, usará ese espacio temporal para almacenar, en el primer octeto, la dirección donde están las constantes, y el resto lo actualiza todo a ceros, quedando:

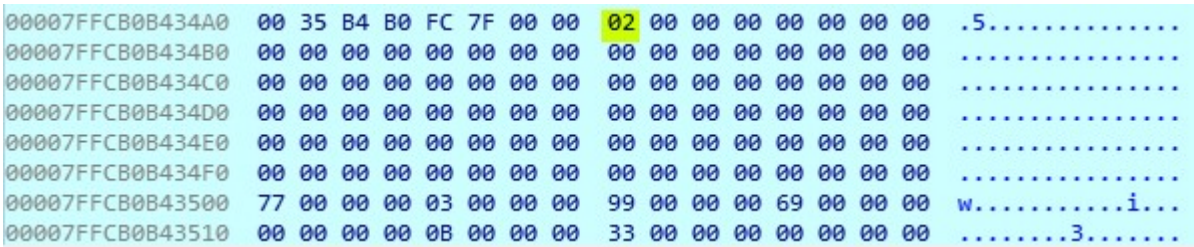


Y después invoca a la función xd::Run, ya solo le pasa un parámetro, el array_temporal_60, porque de ahí ya puede luego sacar la dirección de las constantes:

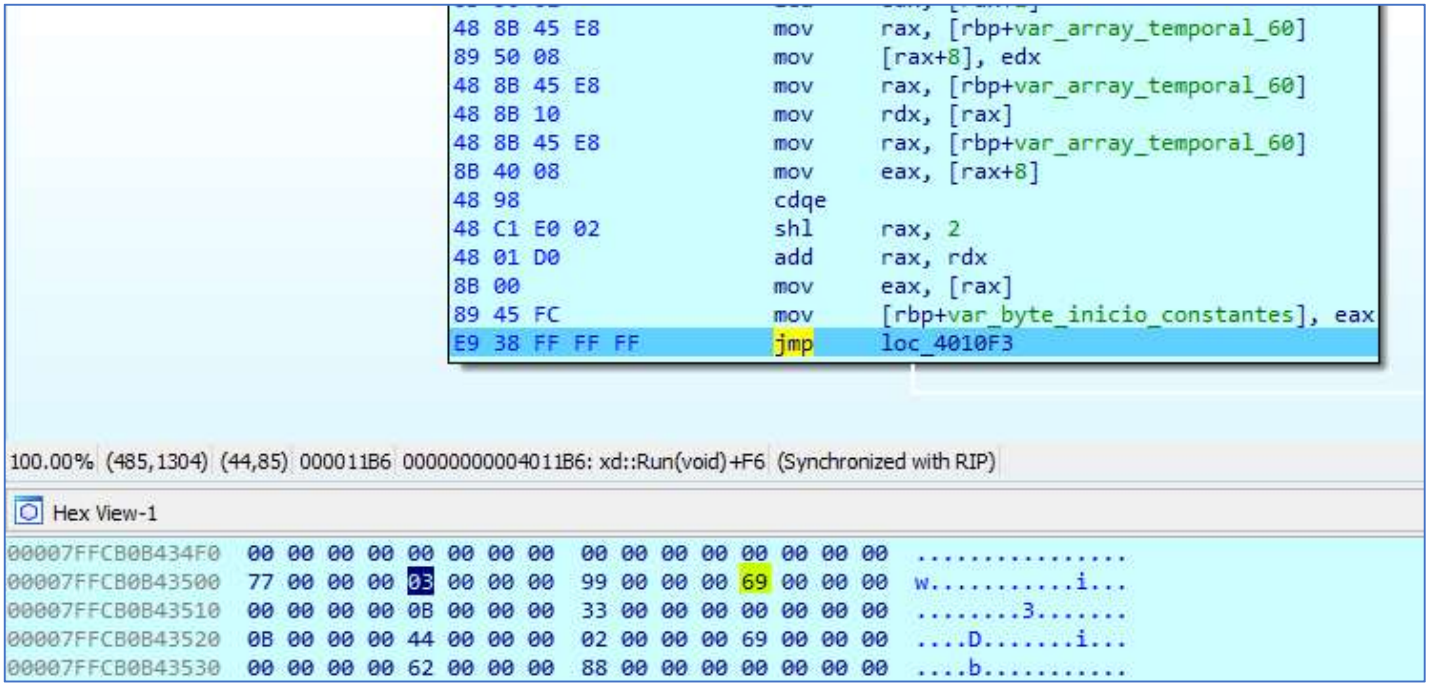


Ahora ya va a ir ejecutando cada una de las funciones de la máquina virtual, según el valor que haya en cada momento.

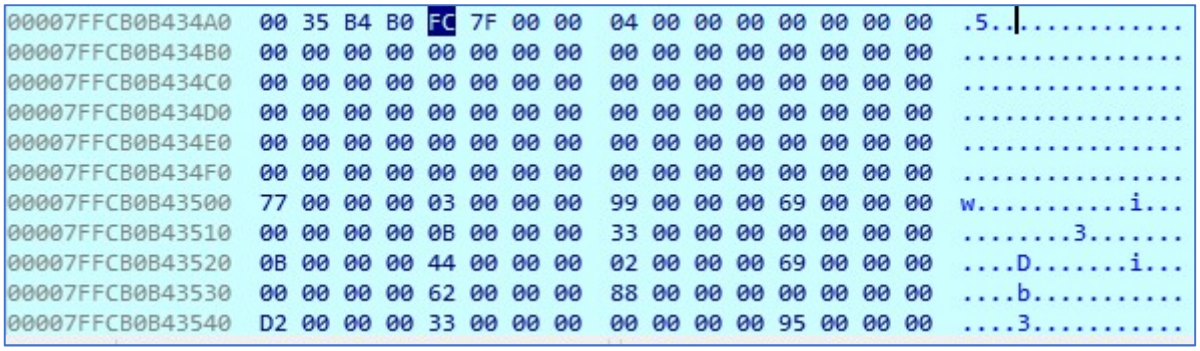
- El primero vemos que tiene un 77, que es el JMP → El resultado de la función es meter un 02 en la zona de almacenamiento temporal. Este valor parece que será siempre el número de índice (cuarteto de bytes) de la instrucción a ejecutar dentro del espacio de constantes:



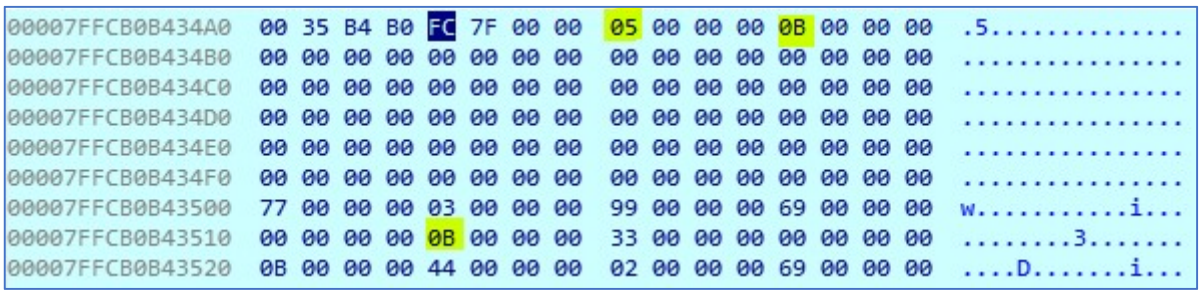
Al final de cada función tiene un código común, que coge el valor que dejó cada opcode de la VM, le suma 1 y lo multiplica por dos (shl 2). Esto se traduce más o menos en, si el valor que dejó es cero, suma 1 y multiplica por 2 para totalizar 4, e ir al siguiente valor de las constantes. Si en este caso, el JMP dejó un 2, pues acaba saltando 3 valores constantes, hasta pasar del 0x77 que era el primero, a 3 más allá, el 0x69:



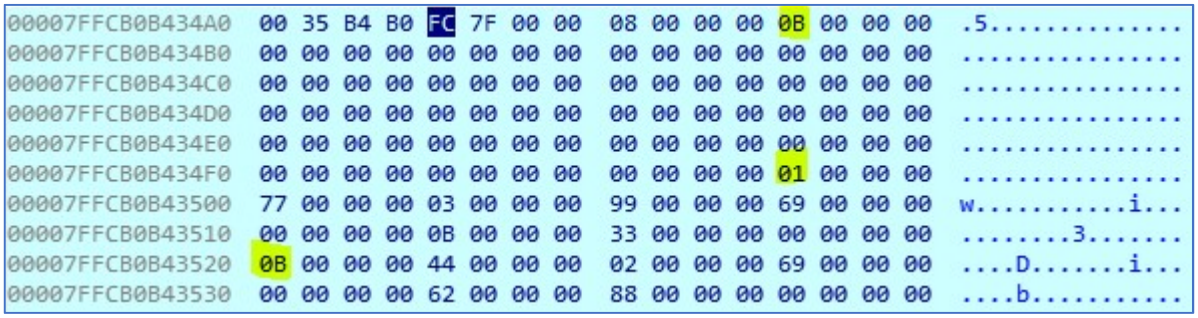
- El 69 es un MOVRV → Mueve el valor del registro que toca al espacio temporal. Así estaba antes de ejecutar:



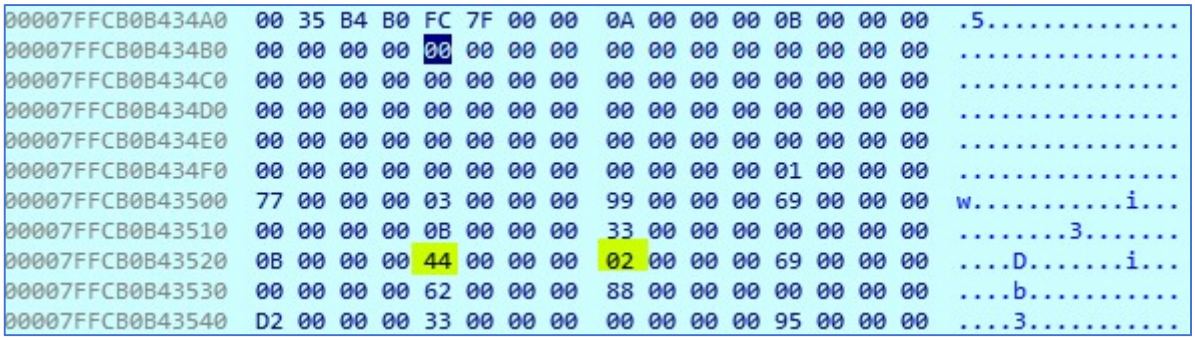
Y así después, vemos como el valor era 0xB y lo ha copiado. Ese 0xB parece que será como una pila para uso temporal, o un registro temporal. Además, el valor del índice anterior pasa del 3 al 5, luego no olvidemos que siempre después de cada función de la VM le suma 1:



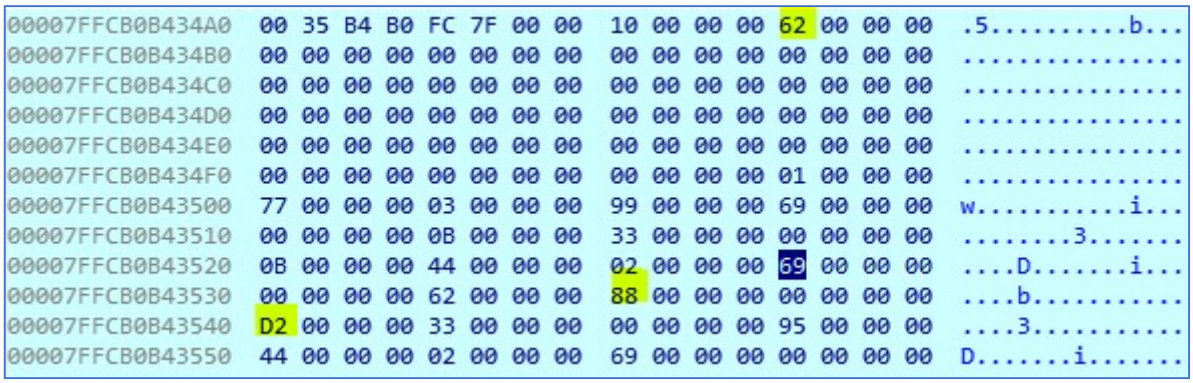
- Valor 33, es un CMP_EQ: compara el valor que tenemos en la pila con el siguiente valor de la tabla de constantes, en este caso era 0xB con 0xB. Como es bueno, el resultado (1) lo mete al final del array temporal.



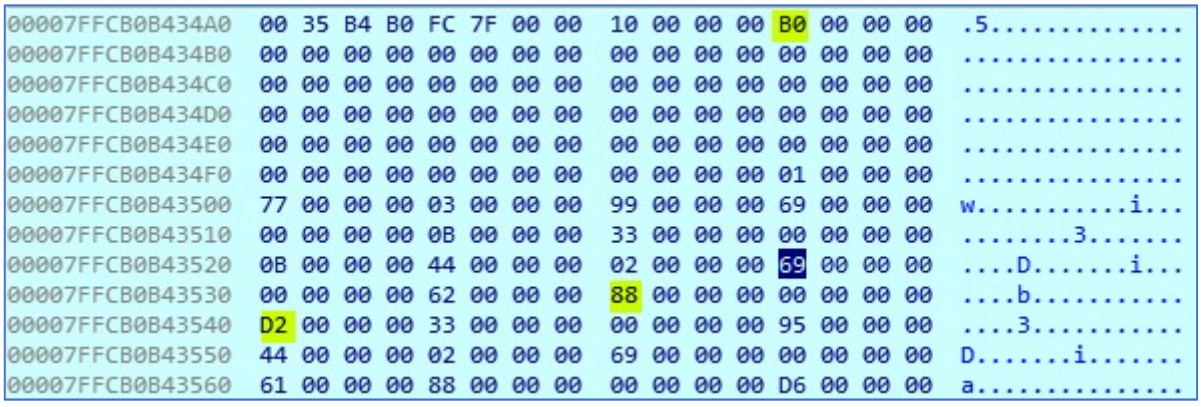
- Valor 44, es un JMP_NEQ, debe ir precedido por supuesto de un CMP anterior: En este caso, si la condición metida al final del array (resultado del CMP previo) fue distinto de 0, no hace nada. Si fue cero salta el numero de posiciones que tiene en el registro (en este caso hubiera sido un 02). Memoria después de la operación:



- Valor 88, es un XOR, coge el valor que había en el registro temporal (un 0x62) y le hace XOR con el valor que va detrás del 88, en este caso un 0xD2. Foto antes:



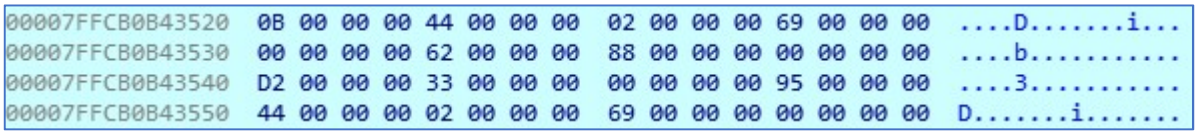
Y la foto después:



Este 62 (una b) que había justo antes método, es el primer carácter que metí en la flag introducida (banderaUAM):

- Valor 55, en este caso es un MOVRR: no llego a utilizarla.

Por tanto, una vez tenemos más o menos claro lo que hace cada función de la máquina virtual, vamos a "ejecutar" su código. Vemos que todas siguen un patrón común:



- 69 00 62 → MOVRV 62 ; Coge el valor 62 (b) y la pasa al registro temporal.
- 88 00 D2 → XOR D2 ; Hace un XOR de D2 con el valor del registro temporal (62)
- 33 00 95 → CMP_EQ ; Compara el resultado anterior con 95
- 44 02 → JMP_NEQ ; Si la comparación fue mala, salta dos registros.

Este patrón se va repitiendo todo el rato para todas las letras que metimos en la cadena de entrada (banderasUAM), en cada caso va haciendo un XOR con un valor diferente, así que vamos uno a uno sacándolos. Además, en este caso, usando la propiedad inversa del XOR, como tengo el valor por el que hace la operación y el resultado que espera, si hago el XOR de estos dos valores me dará el original que necesito.

```
 1 b      62    XOR D2 == 95      --> 47 (G)
 2 a      61    XOR D6 == E6      --> 30 (0)
 3 n      6E    XOR 87 == D3      --> 54 (T)
 4 d      64    XOR EA == B5      --> 5F (_)
 5 e      65    XOR D4 == BC      --> 68 (h)
 6 r      72    XOR 02 == 32      --> 30 (0)
 7 a      61    XOR 1B == 2B      --> 30 (0)
 8 s      73    XOR 09 == 62      --> 6B (k)
 9 U      55    XOR AC == 9D      --> 31 (1)
10 A      41    XOR 10 == 7E      --> 6E (n)
11 M      4D    XOR AA == CD      --> 67 (g)
```

Por tanto, la flag que hay que meter, para que luego todos los XORes y los valores cuadran, sería “G0T_h00k1ng”.

Lo validamos contra el ejecutable y efectivamente nos dice que es correcto:

```
nacho@kali:~/UAM/201910-BolaDrac3$ ./main
Flag: G0T_h00k1ng
Flag correcta! :D
UAM{G0T_h00k1ng}
```

Por tanto, haciendo el MD5 de la flag, nos quedará UAM{7b02cd3d2d3cea80359cf600799413d3}

José Ignacio de Miguel González:

User UAM: nachinho3

Telegram: @jignaciodemiguel